例 6.17　异步复位 D 触发器

```
module as_DFF(d,clr,clk,qout);
  input [7:0] d; input clr,clk; output reg [7:0] qout;
  always @(posedge clk or posedge clr)
    begin
      if (clr) qout<=8'h00;
      else qout<=d;
    end
endmodule
```

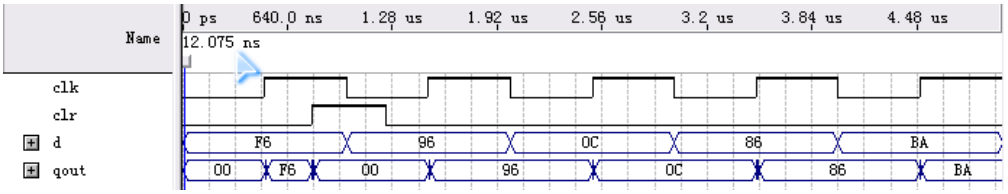



图 6.12　例 6.17 的仿真时序（QuartusII）

比较例 6.16 和例 6.17 可以看到，同步复位必须在时钟 clk 的控制下进行，而异步复位可在任何时候由复位端 clr 控制进行。

锁存器一般是电平敏感型。例 6.18 给出了一个 4 位锁存器的实例。

例 6.18　4 位锁存器

```
module latch_4(d,clr,set,ld,qout);
  input [3:0] d; input clr,set,ld; output reg [3:0] qout;
  always @(*)
    begin
      if (clr) qout<=4'h0;
      else if (set) qout<=4'hf;
      else if (ld) qout<=d;
      else qout<=qout;
    end
endmodule
```

本例中，输入数据为 d，复位信号为 clr，置位信号为 set，锁存信号为 ld，输出为 qout。其仿真时序如图 6.13 所示。

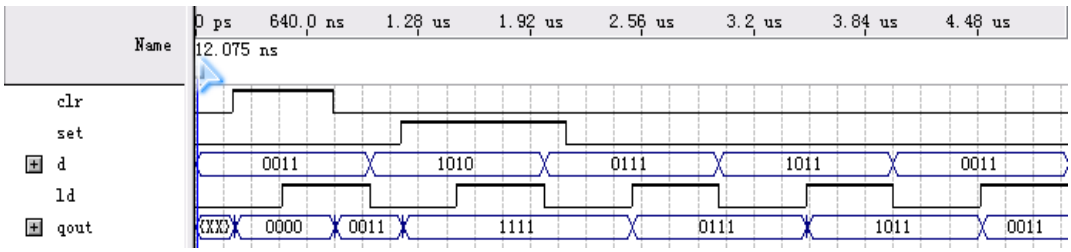


图 6.13　例 6.18 的仿真时序（QuartusII）

请注意，如果 clr 和 set 出现同时为“1”的情况，电路应怎样工作？其工作过程如图 6.14 所示。这是因为根据 if-else 语句的顺序性，clr 获得优先执行权。

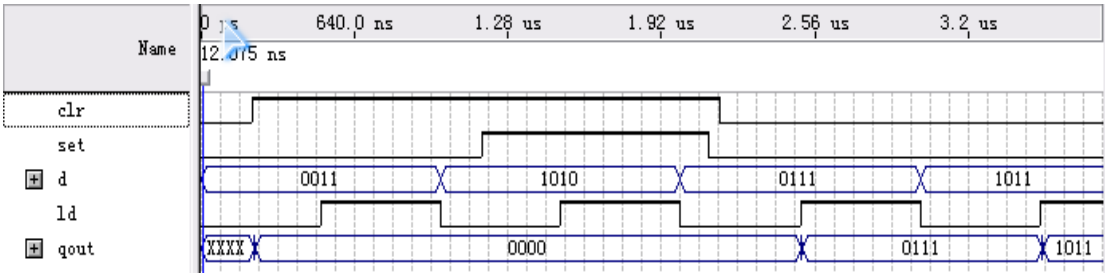



图 6.14　clr 和 set 同时为“1”时的仿真时序（QuartusII）

## 6.2.2　移位寄存器设计

例 6.19 描述了一个一般形式的移位寄存器。其仿真时序如图 6.15 所示。

例 6.19　一般形式的移位寄存器

```
module G_shift_reg (mode,clr,clk,rin,lin,parin,qout);
  parameter size=8 ;
  input [1:0] mode; input clr,clk,rin,lin; input [size-1:0] parin;
  output reg [size-1:0] qout;
  always @(posedge clk or posedge clr)
    begin
      if (clr) qout<= 0;
      else
        case (mode)
          2'b00: qout<={lin,qout[size-1:1]};        //右移
          2'b01: qout<={qout[size-2:0],rin};        //左移
          2'b10: qout<=parin;                       //并行输入
          default: qout<= qout;
        endcase
    end
endmodule
```

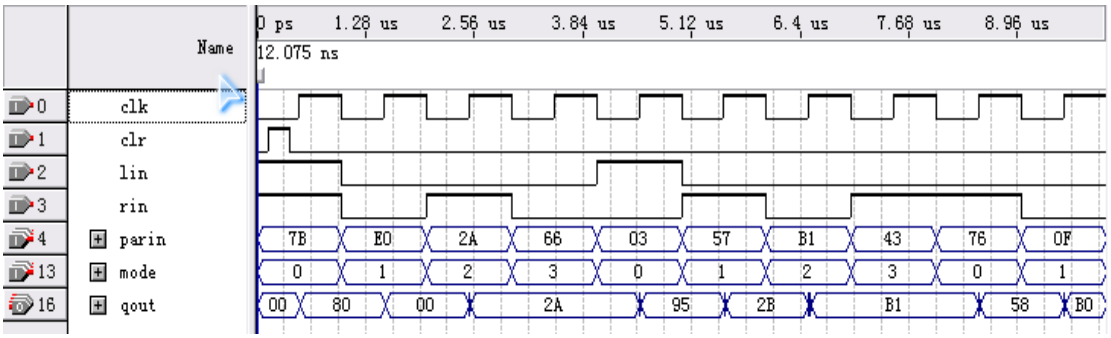



图 6.15　例 6.19 仿真时序（QuartusII）

例 6.20 描述了一个产生 *m* 序列（也称 PN 序列或伪随机序列）的移位寄存器，由 7 个寄存器生成一种 *m* 序列或最大长度移位序列，$m=2^7-1$。产生 *m* 序列的多项式为：$D^7+D^1+D^0$。*m* 序列的初始状态可用一输入 a 控制。

例 6.20　生成 *m* 序列的移位寄存器

```
module pn_g(clk,rst,a,pn);
   input clk;                    //system clock 系统时钟
   input rst;                    //system reset 系统复位
   input [6:0] a;                //shifter initial state 移位初始状态
   output pn;                    //coding output 编码输出
   reg [6:0] register; wire pn;
   always@(posedge clk)
      begin
         if (rst) register<=a;
         else begin register[6:0]<={register[5:0],register[0]^register[6]}; end
                                   //D7+D1+D0
      end
    assign pn = register[6];
endmodule
```

其仿真时序如图 6.16 所示。

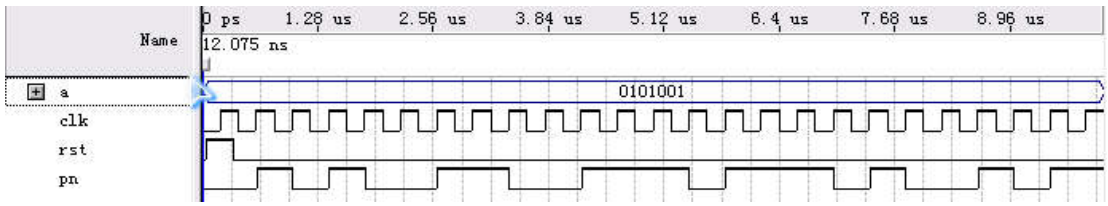



图 6.16　例 6.20 的仿真时序（QuartusII）

### 6.2.3 计数器设计

例 6.21 描述了一个参数化可变模加/减法计数器。其中 din 是预置数，clk、rst、ld、u_d 和 qout 分别是时钟、复位、加载、加/减计数控制和计数输出。其仿真时序如图 6.17 所示。

例 6.21 参数化可变模加/减法计数器

```
module cnt_updown_p(din,clk,rst,ld,u_d,qout);
  parameter size=10;
  input [size-1:0] din; input clk,rst,ld,u_d;
  output reg [size-1:0] qout;
  always@(posedge clk)
    begin
      if (rst) qout<=0;
      else if (ld) qout<=din;
      else if (u_d) qout <=qout+1;
      else qout <=qout-1;
    end
endmodule
```

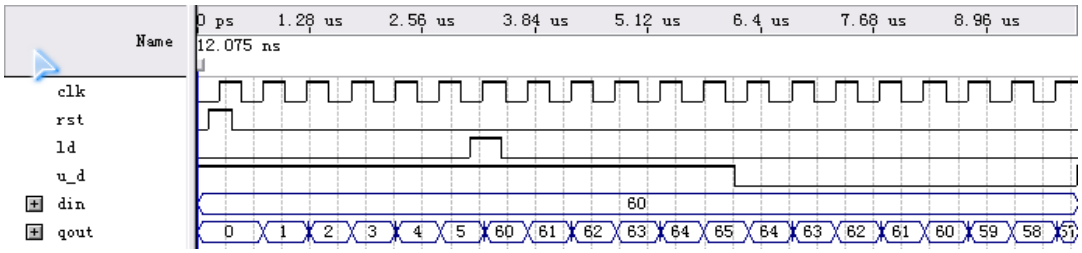



图 6.17 例 6.21 仿真时序（QuartusII）

例 6.22 描述了一个通用参数化 Johnson 计数器。该计数器的计数状态具有相邻计数输出只有一位发生改变的特点。一般 $n$ 位 Johnson 计数器有 $2n$ 个计数状态。例如，$n$ 为 4 位情况下输出为：0000→0001→0011→0111→1111→1110→1100→1000→0000→…

例 6.22 通用参数化 Johnson 计数器

```
module Johnson_p(clk,rst,qout);
  parameter size=10;
  input clk,rst;
  output reg [size-1:0] qout;
  always@(posedge clk)
    begin
      if (rst) qout<=0;
      else qout<={qout[size-2:0], ~qout[size-1]};
```

```
        end
    endmodule
```

其仿真时序如图 6.18 所示。

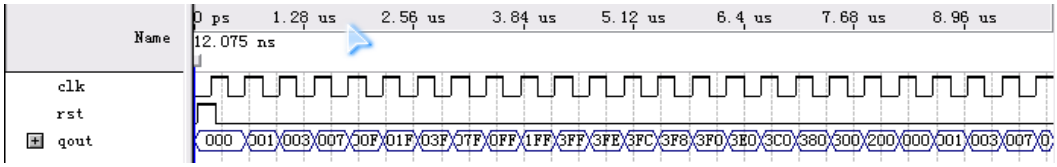



图 6.18　例 6.22 仿真时序（QuartusII）

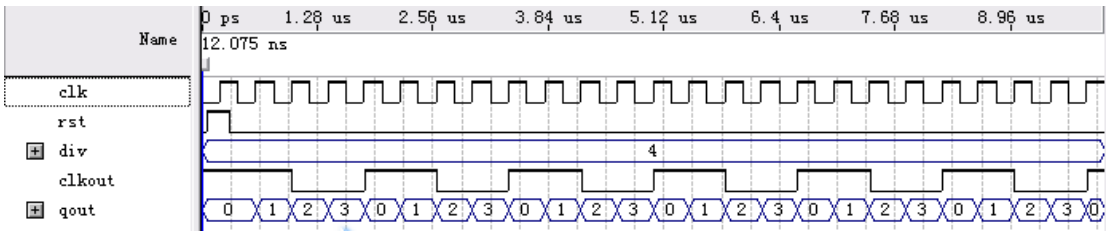
## 6.2.4　分频器设计

在分频比为偶数时，得到占空比为 50%的分频输出波形是很容易的，但在分频比为奇数时，则不能获得占空比为 50%的分频输出。例如：

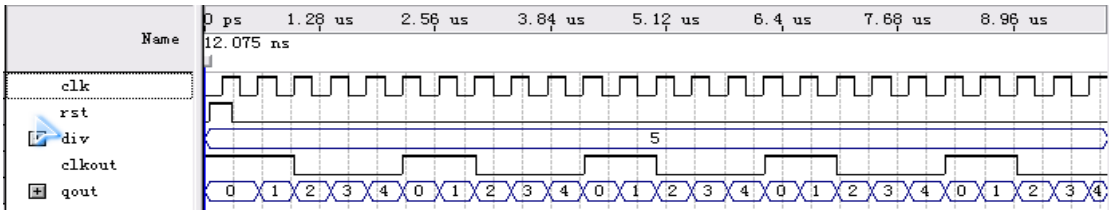
例 6.23　任意倍数计数分频器

```
module div_cnt (clk,rst,div,clkout);
   parameter size=4;
   input clk,rst; input [size-1:0] div;
   output clkout;
   reg [size-1:0] qout;
   always@(posedge clk)
         begin if (rst|| (qout==(div-1))) qout<=0;else qout<=qout+1; end
   assign clkout=(qout<=(div/2-1))?1:0;
endmodule
```

在分频比为 4 和 5 时，其仿真时序如图 6.19（a）和（b）所示。。




（a）分频比为偶数 4 时的仿真时序（QuartusII）




（b）分频比为奇数 5 时的仿真时序（Quartusll）

图 6.19　例 6.23 的仿真时序

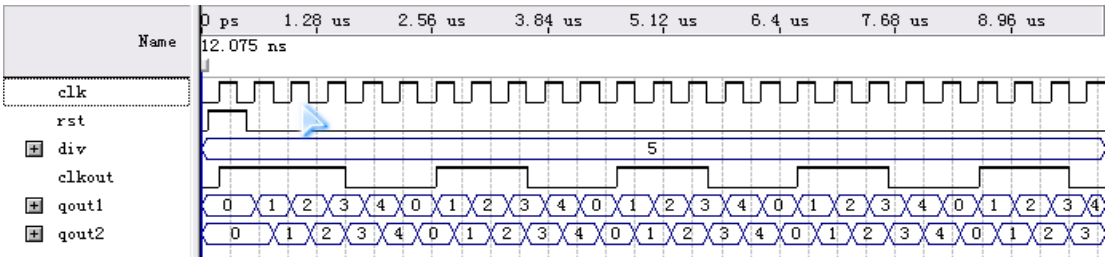
可以看到，分频比为偶数 4 时，可以得到占空比为 50%的输出 clkout，而分频比为奇数 5 时，则得不到占空比为 50%的输出，因此，必须加以改进。

一般在分频比为奇数时，为得到 50%的占空比可使用两个计数器，分别利用时钟的上升沿和下降沿。例如：

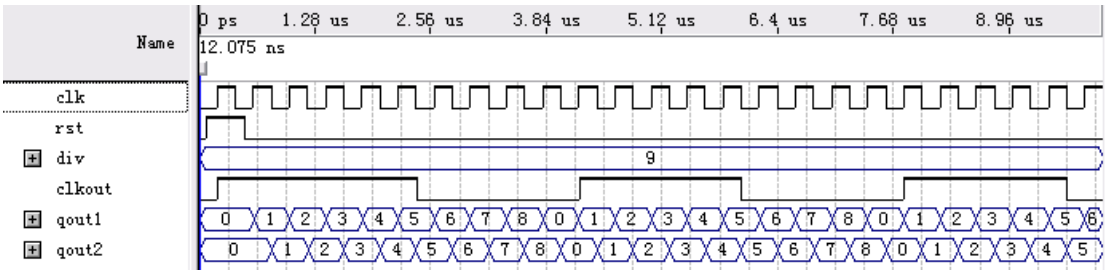
例 6.24　奇数分频器

```
module div_odd (clk,rst,div,clkout);
   parameter size=4;
   input clk,rst; input [size-1:0] div;
   output clkout;
   reg [size-1:0] qout1,qout2; reg clkout1,clkout2;
   always@(posedge clk)
      begin
        if(rst||(qout1==(div-1))) qout1<=0;else qout1<=qout1+1;
        if(qout1<=(div/2-1)) clkout1<=1; else clkout1<=0;
      end
    always@(negedge clk)
       begin
         if (rst||(qout2==(div-1))) qout2<=0;else qout2<=qout2+1;
         if (qout2<=(div/2-1)) clkout2<=1; else clkout2<=0;
       end
    assign clkout=clkout1|| clkout2;
endmodule
```

其仿真时序如图 6.20 所示。




（a）分频比为奇数 5 时的仿真时序（Quartusll）




（b）分频比为奇数 9 时的仿真时序（Quartusll）

图 6.20　例 6.24 的仿真时序

## 6.3　存储器设计

存储器的类型一般有：RAM、ROM、FIFO 等。它们的设计可以使用 EDA 软件的参数化模块库，也可以使用硬件描述语言。

### 6.3.1　基于 Verilog HDL 的存储器设计

#### 1. 基于 Verilog HDL 的 ROM 设计

例 6.25 给出了一个基于 Verilog HDL 的 16×8 ROM 设计实例。其仿真时序如图 6.21 所示。

例 6.25　16×8 的 ROM

```
module rom_16x8(read,addr,data);
  input read; input [3:0] addr;
  output reg [7:0] data;
  always@(read or addr)
```

```
        begin
          if (read)
            begin
              case(addr)
                4'd0:data<=8'd2; 4'd1:data<=8'd4; 4'd2:data<=8'd5;
                4'd3:data<=8'd1;4'd4:data<=8'd3; 4'd5:data<=8'd56;
                4'd6:data<=8'd122; 4'd7:data<=8'd200;4'd8:data<=8'd124;
                4'd9:data<=8'd232; 4'd10:data<=8'd241; 4'd11:data<=8'd252;
                4'd12:data<=8'd21; 4'd13:data<=8'd32; 4'd14:data<=8'd9;
                4'd15:data<=8'd11;
              endcase
            end
          else data<=8'dz;
        end
    endmodule
```

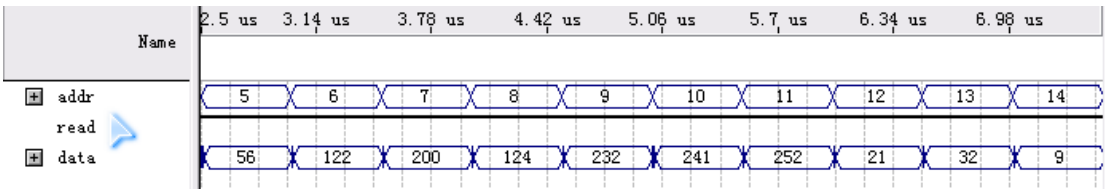



图 6.21　例 6.25 的仿真时序（QuartusII）

## 2. 基于 Verilog HDL 的 RAM 的设计

例 6.26 给出了一个基于 Verilog HDL 的 16×8 ROM 设计实例。其仿真时序如图 6.22 所示。

例 6.26　16×8 的 RAM

```
module ram_16x8(clk,cs,rw,addr,din,dout);
  input clk,cs,rw; input [3:0] addr; input [7:0] din;
  output reg [7:0] dout;
  reg [7:0] ramb [15:0];
    always@(posedge clk)
      begin
        if (cs)
          begin
            if (rw) dout<=ramb[addr];
             else ramb[addr]<=din;
```

```
        end
      else dout<=8'dz;
    end
endmodule
```

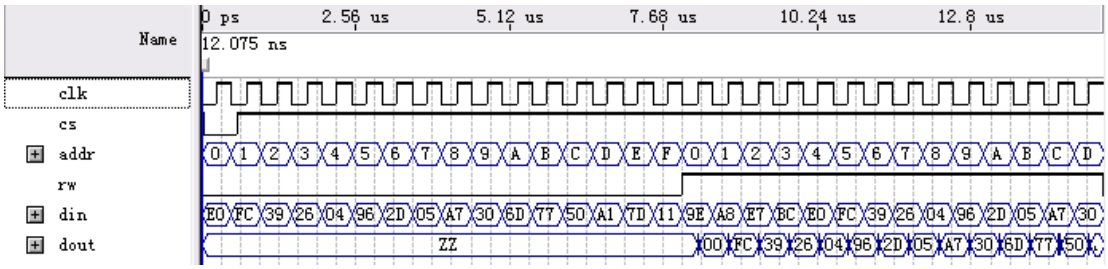



图 6.22　例 6.26 的仿真时序（QuartusII）

### 6.3.2　基于 LPM 函数的存储器设计

存储器也可以通过 MegaWizard Plug-in Manager 进行设计。其具体方法已在第 3 章详细介绍过，此处不再赘述。

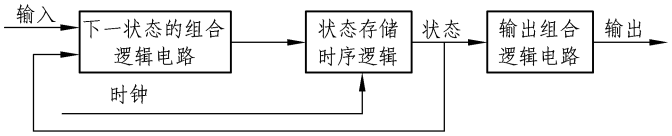
## 6.4　有限状态机设计

### 6.4.1　有限状态机的基本结构

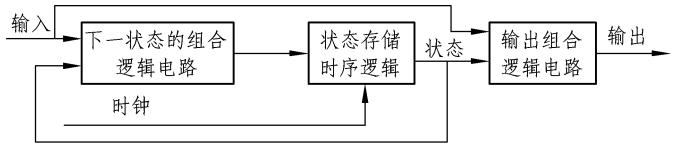
有限状态机（FSM，Finite State Machine）在数字电路设计，特别是需要对高速器件进行控制的设计中起着重要的作用。其优点是：控制灵活，结构简单，可靠性高，实现速度快，等等。

有限状态机根据是否受同一个时钟控制，分为同步和异步两种类型，即受同一个时钟控制的有限状态机称为同步状态机，否则，称为异步状态机。这里主要介绍同步状态机。

一般同步状态机的基本结构如图 6.23 所示，有两种类型。可以看到，状态机的输出仅与当前状态有关的，称为摩尔（Moore）型状态机；状态机的输出不仅与当前状态有关还与当前输入有关的，称为米里（Mealy）型状态机。




（a）摩尔（Moore）型状态机




（b）米里（Mealy）型状态机

图 6.23　同步状态机的基本结构

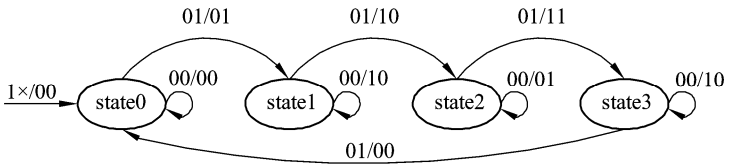
由图 6.23 还可以看到，不论是摩尔型还是米里型状态机，均含有下一状态的组合输入电路。该电路根据当前输入和当前状态决定状态存储时序逻辑电路的输入，并据此决定下一个状态的情况。输出组合逻辑电路则根据当前状态、当前输入（米里型）决定有限状态机的输出。

总之，有限状态机应实现如下功能：根据当前状态和当前输入产生其内部状态转换的条件，根据条件实现内部状态的转换，根据当前的状态和输入（米里型）决定当前输出。

## 6.4.2　有限状态机的描述及设计要点

### 1. 有限状态机的描述

有限状态机的描述一般包括：状态转移图、状态转移表及状态流程图。状态转移图利用有向图表示状态转移的情况，其中的节点表示状态机的状态，有方向的连线表示状态转移的方向、输入条件和输出结果。如图 6.24 所示为一个米里型有限状态机的状态转移图。




状态图有向连线：输入/输出
AB/CD
A：输入 A
B：输入 B
CD：两位输出 C、D

图 6.24　米里型有限状态机的状态转移图

由图 6.24 可以看到：这个有限状态机包括 4 个状态，分别为 state0、state1、state2 及 state3；状态机的输入有两个，一个是 A，另一个是 B；输出是一个两位（bit）为 C、D 的变量。当输入 A 为“1”时，无论输入 B 为何值，状态均要回到 state0，输出为“00”。在输入 A 为“0”的情况下，当输入 B 为“1”时，状态将发生转移并产生相应输出；当输入 B 为“0”时，状态将不发生转移，即维持在原状态并产生相应输出。

需要注意的是：一般米里型状态机的状态转移图中，输入与输出标在表示状态转移的有